# A planner-based approach to automated processing and tracking of mission data

**Keith Golden**
NASA Ames Research Center
M/S 269-2
Moffett Field, CA 94035-1000
kgolden@ptolemy.arc.nasa.gov

**Abstract**

Managing the terabytes of data gathered by satellites and other sensors is one of NASA's greatest challenges. Finding desired data products is difficult, especially years after a mission is over. Processing and delivering data to scientists and the public in a timely manner can be a challenge, even with current methods of automation.

Data management includes both data processing and data tracking. We address the data management problem by casting it as an AI planning problem. Actions are data-processing commands, plans are dataflow programs and goals are metadata descriptions of desired data products. Data processing is simply plan generation and execution, and a key component of data tracking is inferring the effects of a dataflow plan that is known to have produced a given data product.

We introduce a tractable approach to planning for data processing, and we describe a particular embodiment of this approach, called ADLIM, for Action Description Language for Information Manipulation. We discuss the connection between data processing and information integration and why action representations for information integration are not sufficient for data-processing domains. We also discuss how to gather information within a data-processing framework, and show how ADLIM metadata expressions are a generalization of Local Completeness.

## 1 Introduction

### 1.1 The NASA data management problem

Managing the terabytes of data gathered by satellites and other sensors is one of NASA's greatest challenges. Satellites, unmanned spacecraft, planetary rovers and observatories, for all their complexity, can all be viewed as remote sensors; their sole purpose is to gather data, which are then processed, delivered to the scientific community and the public, and archived. The data management problem is especially acute in the Earth Sciences, where the data sets are large and diverse, and there is an increasing demand for real-time data processing in support of a wide range of scientific tasks. Real-time data processing in support of novel science goals is not feasible with the current approaches, which typically rely on (1) human-generated scripts to perform routine, expected operations, (2) manual data processing to handle special cases and (3) filenames and headers to store metadata. Novel data requests must either be processed manually or will require new data-processing scripts, which take time to write and debug.

Furthermore, there is a tremendous problem in *data tracking*: keeping track of what data exist, what the data represent, and where they are stored. Current approaches leave a significant portion of such vital metadata unrecorded, or recorded in ways that that greatly limit their accessibility, such as the metadata implicit in filenames and in the directory hierarchy, making it difficult to locate data years after they have been produced, especially by those who were not involved in the production.

Figure 1 shows a typical data processing operation. Arcs in the figure represent dataflow. Data processing consists of constructing and executing such dataflow programs to produce a desired result. Data tracking consists of deriving a metadata description of data products produced by such programs, and storing it in a database, to facilitate later data searches.

We address the data management problem by casting it as an AI planning problem. Actions are data-processing commands, plans are dataflow programs like Figure 1, and goals are metadata descriptions of desired data products. Data processing is simply plan generation and execution, and a key component of data tracking is inferring the effects of plans known to have produced given data. We introduce a new action language for data management domains, called ADILM.

Although data tracking is the easier problem computationally, it places more demands on the representation of data and the actions that manipulate data, since the system must be able to generate correct and useful metadata descriptions for the output of any dataflow program, regardless of how it was generated. For example, Collage [14] and MVP [1] are both planners that have been used to automate data manipulation (in particular, image processing), but they do not automate data tracking, and in fact neither uses a representation that is suitable for metadata generation. Both use an HTN representation, which allows them to avoid pro-

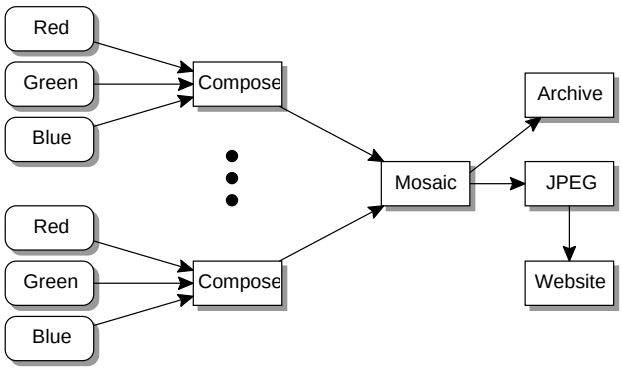


Figure 1: A dataflow plan. First, separate monochrome images taken through red, green and blue filters are combined to form a color image. Then, these images are tiled to form a mosaic. Finally, the full resolution image is archived and a JPEG-compressed version is stored on a public website.

viding a detailed causal theory of data processing or data goals, focusing more on the procedures for data processing. However, it is precisely this detailed causal theory that is needed to determine the effects of arbitrary data-processing plans.

The rest of this paper describes an approach to planning for data processing. Section 2 discusses the connection between data processing and information integration and explains why approaches used for information integration are inappropriate for data processing domains. Section 3 discusses how a causal representation of data-producing actions can facilitate reasoning about data-processing plans. Section 4 discusses "filter" actions that transform data. Section 5 discusses the metadata representation, of which action effects are a special case. Section 6 discusses how to gather information in a data-manipulation framework and shows that the ADLIM metadata representation is a generalization of Local Completeness. Section 7 discusses how to reason about data-processing plans.

## 2 Data manipulation vs Information Integration

There has been substantial work in the area of planning for information gathering and information integration, including [13, 16, 12, 7, 10, 2, 11] and many others. This work can be broadly characterized as extracting information from a number of data sources and combining the results in order to answer a user query. For example, if a user requests a listing of all movies currently showing in San Francisco starring Jackie Chan, this query could be answered by going to a web site, such `imdb.com`, to get a list of movies starring Jackie Chan and another site, (such as `sfgate.com`) to find out which of those movies are showing in San Francisco. An essential feature of this and other information-integration problems is that the interesting contents of the data sources can be completely represented in a simple logical language, such as SQL or Datalog. Aspects of the HTML documents for these sites that are not captured by the logical description are not relevant to the problem of finding movie listings.

> *In information integration tasks, the end product is always information; once information is extracted from data, the data can be discarded, since all subsequent operations are on a logical representation of the information.*

Data manipulation concerns tasks like image processing and converting from one data format to another. Data files are processed by one or more programs (filters) to produce new data files. Although it is possible to describe the information contained in the data in some logical language, these descriptions do not completely characterize the data.

> *In data manipulation tasks, the end product is data; information may or may not be extracted from data files, and most operations act directly on the data.*

## 3 Two kinds of sensors

> ***data**: information output by a sensing device or organ that includes both useful and irrelevant or redundant information and must be processed to be meaningful*
>
> — Merriam-Webster OnLine (http://m-w.com)

Since data come from sensors, a good place to start the discussion of representing data is how to represent sensors. There are two types of sensors. One is a "sense organ," such as an eye, which feeds information directly into the brain. The other is a "sensing device," such as a Geiger counter, which produces data that must in turn be sensed and interpreted.

Planners for information gathering or integration represent sensors as sense organs. For example, to represent that executing the action `ls /bin` reveals the name of every file in `/bin`, both the UWL [6] and SADL [8] action languages used by the Internet Softbot [4] use the annotation **observe**, where **observe**(name($f$, $n$)) means the softbot will observe that the name of file $f$ is $n$. At execution time, the appropriate values of $f$ and $n$ will be determined, and the corresponding propositions will be inserted into the softbot's knowledge base. This is the most direct and intuitive way to represent a sensor. What this encoding actually represents, however, is not `ls`, but the combination of `ls` with a program (called a *wrapper*) to read in and interpret its output. The wrapper itself is a "black box" to the planner; the planner "knows" what information is contained in the output of `ls`, but knows nothing of how that information is encoded. In fact, the output is not even mentioned anywhere in the action description. Thus, this representation is not suitable for data management domains, in which the data output itself is of interest.

A sensing device, in contrast, produces an output or result that depends on the state of the world. This output may be perceived and interpreted to obtain information about the world, but the interpretation depends on knowing how the output was produced. This kind of indirect sensing, or *testing* [17], is exemplified by Moore's litmus paper example [18]. The main idea behind testing is to exploit actions with conditional effects to obtain information about properties of the world that are not directly perceptible (such as the acidity of a solution or the amount of radiation emitted by some source). This strategy only works if the outcome of the conditional effect (such as the color of the litmus paper or the frequency of clicks from the Geiger counter) is directly or indirectly perceptible. Thus, testing must always bottom out in some sense organ. In the litmus paper example, it bottoms out when the photons reflected from the paper strike the retina, producing the sensation of *redness* or *blueness*. The agent then reasons backward from its causal theory to determine what the redness or blueness says about the acidity of the solution.

The softbot analogue of the retina is working memory; in ADLIM, direct perception occurs when the output of a sensor is loaded into working memory. Directly perceptible properties are properties of the output that can be computed without resorting to any additional information about the state of the outside world. Anything else must be inferred from the known causal relationship between the state of the world and the data contents.

For example, consider the ADLIM representation of the "piped" version of `ls`, where the output is directed to the input of some other command (as opposed to the screen). Assume that the output of `ls` is designated by the variable *out*, and that `ls` has no preconditions. The effect of `ls` would be written as

$\forall f$: file, $n$: filename. (parent.dir($f$) = /bin $\wedge$ $n$ = name($f$)) $\rightarrow$ containsLine($n$, *out*)

This translates to "For each file in directory `/bin`, there is a line in the output that is equal to the name of the file." The $\rightarrow$ is used to indicate a conditional effect. The expression on the left hand side (LHS) of the $\rightarrow$ refers specifies the conditions before the action is executed needed for the expression on the right hand side (RHS) to become true after the action is executed. The predicate containsLine ($s_1$, $s_2$) means that string $s_1$ appears in string $s_2$, delimited by newlines. The truth value of this predicate can be computed given only the strings $s_1$ and $s_2$, thus satisfying the requirements stated above for direct observability. The function parent.dir($f$), on the other hand, cannot be directly perceived, but once the softbot knows what strings appear in the output *out*, it can infer what files are contained in the directory `/bin`.

There is a qualitative difference between predicates and functions like containsLine, which are directly perceptible and those like parent.dir, which are not:

- The function parent.dir is a *fluent*. That is, its value depends on the state of the world. Knowing the value of $f$ is not enough to know the value of parent.dir($f$); it can be different in different hypothetical states of the world and can change from time to time.
- The relation containsLine is *static*. Its value depends only on the values of its arguments, and no change to the world can change its value. "Sensing" its value is reduced to loading its arguments into working memory and performing a simple computation. There are no preconditions or side-effects to such a computation, so an explicit sensing action is unnecessary; instead, we can associate with each static predicate or function a procedure for determining its value. Static predicates of this form are sometimes called *facts*.

Comparing this approach to the information-integration approach reveals that the computation performed in reading in the output of a data-producing action, executing procedures to determine the values of static predicates and inferring the values of fluents is exactly what a wrapper does. We've changed the wrapper from a "black box" to a "white box," which is integrated into the reasoning process of the planner itself. This change in representation gives the planner the ability to reason about actions that manipulate data, which is not possible given the "sense organ" representation of information-producing actions.

The difference between this approach and the work in testing [17] is one of simplicity and tractability. We only allow data-producing actions to be used as sensors, so we exclude, for example, trying to open a door in order to find out if it is locked. We also disallow data sources that provide disjunctive information, and our representation of world knowledge is based on three-valued logic, rather than possible worlds. As a consequence, not every conceivable data-management problem can be solved using this representation, but those that can be solved can be solved using straightforward planning techniques.

## 4 Filters and other actions

**Filters** are actions that transform data. A filter has one or more inputs and one or more outputs, and the outputs depend on the inputs in some way, as specified in the action effect description. It does not modify its inputs in any way, and the outputs are always new objects. Furthermore, a filter has no side effects. As with data-producing actions, the effects of filters are specified using conditional effects. However, since filters don't produce information, the LHS cannot refer to the state of the world, but only to the input data. Thus, both the LHS and the RHS of effects are specified with static predicates.

For example, the Unix `grep` command, when used as a filter, outputs the lines of text appearing in its input that contain text matching a given regular expression. For example, "`grep .ps$`" outputs all strings from its input ending in "`.ps`":

$\forall s$: string. (containsLine($s$,*input*) $\wedge$ matches($s$, "`.ps$`")) $\rightarrow$ containsLine($s$,*output*)

Given this description, and that of `ls`, it is easy to see that the output of "`ls /papers | grep .ps$`" (directing the output of `ls /bin` to the input of `grep .ps$`) will contain the names of all files in `/bin` that end in `.ps`. See Figure 2 for a full action description of a filter.

**Data delivery** actions take one or more inputs and change the state of the world to produce some physical embodiment of the data. The LHS refers to the data, and so is expressed using static predicates. The RHS refers to the world, and so is expressed using fluents.

**Data mining** actions extract features that are latent in data but not readily apparent. Such features can be represented using "imperceptible" static predicates, and data mining actions map from imperceptible to perceptible predicates.

**Other actions** are also possible, including ordinary causal actions and "data-producing" actions that don't actually reveal information about the world. Here's an action that just creates a value map (monochrome image) of a specified size and fills it with a specified value.

```
action   makeConstant(c: pixelValue,
                      width, height: natural)
output   MCout: valuemap
forall   x, y: natural
effect   xSize(MCout) = width ∧
         ySize(MCout) = height ∧
         ((x < width ∧ y < height) →
              value(x, y, MCout) = c)
```

## 5 Representing metadata

The effect description of `ls` from Section 3 can be regarded as a *metadata* description of the output of `ls`. It contains two vital components:

1. the information contents of the data (represented using fluents)
2. how the information is encoded in the data (represented using static predicates)

The $\rightarrow$ relates the two. The symbol $\rightarrow$ does not merely denote implication, since it involves a temporal element as well as a logical element. When it appears in the form of a conditional effect, the $\rightarrow$ relates the truth value of the LHS *before the action is executed* with the truth value of the RHS *after the action is executed*. However, since the expression on the RHS is static, the only time point of interest is that of the LHS. Thus, another component of metadata is:

3. what time the information pertains to

For data-producing actions, this time is whenever the action is executed, so there is no need to state it explicitly in the action description, but once the output has been produced, it is necessary to keep track of what time it pertains to. For example, if a softbot is to produce nightly backups, the night a given backup was made should be recorded.

For data *goals*, the time that the information pertains to is also of interest. As discussed in [8], failure to specify the time for which information is requested is the reason why a goal of "tell me the color of the door" could achieved by painting the door blue and then answering "blue." If instead we ask "what is the *current* color of the door," painting the door only obscures that information and does nothing to answer the question. The syntax of data goals is the same as that of data-producing effects, except that goals can refer explicitly to the time of interest for fluents on the LHS, using an extra, optional argument for each fluent. For example, to refer to the color of the door at 9am today, we can write $c$ = color(`door`, 9:00)). If no time is specified, it is assumed to be whenever the goal is given. If a future time is specified, this is interpreted as a request to schedule a data-gathering operation to be carried out when that time arrives.

Another difference between data-producing actions and data goals is that the goals must specify what is to be done with the data, such as the pathname of the file where the data should be put. So another component of metadata is

4. where the data reside (represented using fluents)

Although the location of the data is specified as a fluent, no time is given; for goals, it is assumed to be "as soon as possible"; for metadata in the agent's knowledge base, it is assumed to be "right now," since anything else would amount to incorrect knowledge.

### 5.1 Limited completeness assumption

For fluents in action effects, we make the STRIPS assumption. For static predicates, the STRIPS assumption is too restrictive, because there are many properties of data files, and it would be impossible and inappropriate to list them all for every file produced. For example, containsLine technically describes any files that contain the newline character, including most binary files, but it is really only meaningful for text files. On the other hand, referring to individual bits is appropriate for some binary files, but would be inappropriate for text files, even though all files contain bits. Actions should only describe their outputs in terms of the properties that are meaningful.

For static predicates, we make a limited form of the STRIPS assumption. We require that if any predicate is used in the RHS of any effect or metadata description, then the description must be complete with respect to that predicate. E.g., the output of `ls /bin` cannot contain any lines of text that are not the names of files in `/bin`. Any static property that does not appear in the RHS of a metadata description will be *unknown*.

This assumption applies to metadata knowledge as well as action effects, but not to metadata goals. To specify goals that are "complete" in the above sense, one can use the notation $\overset{\Leftrightarrow}{\rightarrow}$, where $A \overset{\Leftrightarrow}{\rightarrow} B$ is equivalent to $(A \rightarrow B) \wedge (\neg A \rightarrow \neg B)$

### 5.2 Constraints

Many data processing domains require sophisticated constraint reasoning in order to select parameters appropriately [1, 14]. ADLIM provides "built-in" constraints, such as inequality. Additionally, all of the procedures for evaluating static predicates are implemented as constraints. That is, the set of possible values for each argument is restricted based on the other arguments. It is useful to know what arguments will be determined if singleton values are provided for other arguments. We use the predicate bound to represent that the value(s) of an argument are determined. For example, we can write

$$\mathsf{containsLine}(s, \mathit{data}) \wedge \mathsf{bound}(\mathit{out}) \rightarrow \mathsf{bound}(s)$$

to represent that once the output *out* is known, every line of text in *data* can be determined. By definition, the return value of a function (or the boolean value of a relation) will be bound if all its arguments are bound. Additionally, by convention, the last argument of a static predicate is the data being described. If that argument is bound then all other arguments will be bound, as in the above example. This can, however, be overridden to capture *binding patterns*, which require particular arguments to be specified. For example, consider the predicate contains(*s, data*), meaning *data* contains *s* as a substring. It is *possible* to list all substrings of *data* once the value of *data* is known, but it would not be practical.

Additionally, unary constraints can be associated with types. For example, a Unix filename can be any non-empty string that does not contain the character "/". This constraint can be represented by the regular expression "`~[/]+`". Similarly, a natural (number) is an integer whose value is greater than zero, which is represented by an inequality constraint.

### 5.3 Example

Suppose `plot` is a grayscale image (valuemap) corresponding to an elevation map of the San Francisco Bay Area. Let xProj and yProj be linear functions mapping the x, y coordinates of the image to the corresponding longitude, latitude. Let hProj be a linear function mapping elevation to pixel values in the image, where lower (blacker) values correspond to lower elevations. Let elevation(*x, y, w, h*) be a fluent function returning the average elevation over the *w* by *h* square centered at *x, y*. The metadata description of `plot` would be

$$\begin{aligned}&\mathsf{xSize}(\texttt{plot}) = \mathrm{XMAX} \wedge \mathsf{ySize}(\texttt{plot}) = \mathrm{YMAX} \wedge\\ &\forall x, y\text{: natural, } el\text{: real.}\\ &\quad (x < \mathrm{XMAX} \wedge y < \mathrm{YMAX} \wedge\\ &\quad\ el{=}\mathsf{elevation}(\mathsf{xProj}(x),\mathsf{yProj}(y),\mathrm{XRES},\mathrm{YRES})\\ &\qquad \rightarrow \mathsf{value}(x, y, \texttt{plot}) = \mathsf{hProj}(el)\end{aligned}$$

where words in ALL CAPS are constants. We use the following shorthand notation in this example and the rest of the paper. "f(g($x$))" in the RHS is equivalent to "($y$ = g($x$)) → f($y$)" and "f($x$) = g($x$)" in the RHS is equivalent to "($y$ = g($x$)) → (f($x$) = $y$)." E.g., "value($x$, $y$, `plot`) = hProj($el$)" is a shorthand for "($hp$ = hProj($el$)) → value($x$, $y$, plot) = $hp$." Note that this description quantifies over the pixels in the image. It would almost never be desirable to extract the information associated with each pixel, but it is useful to describe the file in terms of what each pixel represents, since image-processing programs typically operate on pixels.

## 6 Information gathering

Most data-processing plans pass data directly from one action to another, without any requirement for the agent to "know" the contents. However, there are cases in which the agent must explicitly gather information in support of planning, to determine the value of a parameter to an action or to make a decision.

In SADL, it is straightforward for an agent to determine what it "knows" as a result of executing actions, since these facts are expressed directly as **observe** effects and are inserted into the knowledge base. In ADLIM, information gathering requires the following steps:

1. formulate an information goal
2. construct a plan to achieve the goal, by finding or creating a file that contains the desired information.
3. execute the plan
4. extract the information from the resulting data.

### 6.1 Information goals

An information goal is just like a data goal except that the format of the data is not of interest. All that matters is that the desired attributes can be extracted from the data. This can be assured using the predicate bound, discussed in Section 5. To obtain the names of all (and only) files in directory `/bin`, we can write

$$\begin{aligned}&(\mathsf{parent.dir}(f) = \texttt{/bin} \wedge n = \mathsf{name}(f))\\ &\qquad \overset{\Leftrightarrow}{\rightarrow} \mathsf{bound}(n)\end{aligned}$$

Since we know that an "effect" of evaluating containsLine is

$$\begin{aligned}&(\mathsf{containsLine}(s, \mathit{data}) \wedge \mathsf{bound}(\mathit{data}))\\ &\qquad \rightarrow \mathsf{bound}(s),\end{aligned}$$

the goal bound($n$) can be satisfied by containsLine($n$, *out*) ∧ bound(*out*), giving the new goal

$$\begin{aligned}&(\neg\mathsf{bound}(n) \wedge \mathsf{parent.dir}(f) = \texttt{/bin} \wedge\\ &n = \mathsf{name}(f)) \overset{\Leftrightarrow}{\rightarrow} (\mathsf{containsLine}(n, \mathit{out}) \wedge\\ &\qquad \mathsf{bound}(\mathit{out})),\end{aligned}$$

where the ¬bound($n$) is used to satisfy the "only if" part of the goal. The containsLine($n$, *out*) can be satisfied by `ls /bin`, the bound(*out*) can be satisfied by reading the output of `ls` into working memory, and ¬bound($n$) is true before `ls` is executed, so executing `ls /bin`, loading *out* into working memory then querying for the possible values of $n$ will reveal the names of all files in `/bin`.

The LHS in information goals must be conjunctive. As mentioned in Section 3, ADLIM cannot be used to represent knowledge about disjunctions.

### 6.2 Local completeness

Given the above restriction, an information goal is equivalent to a Local Closed-World (LCW) [3, 5] or Local Completeness (LC) [15, 7] statement, which expresses the fact that an agent or an information source has complete information about some *locale*, such as the files in a directory. Consider an ADLIM formula of the form

$$\forall x_1, \ldots x_n.\ \Phi(x_1, ..., x_n) \stackrel{\Leftrightarrow}{\rightarrow} (\mathsf{bound}(x_1) \wedge \ldots \wedge \mathsf{bound}(x_n))$$

where $\Phi(x_1, ..., x_n)$ is a conjunctive formula containing only constants and the variables $x_1, \ldots x_n$. If an agent knows all values of $x_1, \ldots x_n$, then it will know all ground instances of $\Phi(x_1, ..., x_n)$ that are true, and thus will be able to determine, for any ground instance, whether it is true or false. That is exactly the definition of LCW [3].

LCW was generalized by [15, 7] to support *constraints* that restrict the set over which an information source is complete, without returning more information about the set. The simpler and more general representation is the LC representation in [7], where a relation is a constraint if it contains a variable whose value is not returned in answer to a query. This can easily be represented in ADLIM. by variables $x$ in the LHS for which there is no $\mathsf{bound}(x)$ on the RHS.

For example, to represent the goal of knowing the names of all files in `/bin` larger than 1 gigabyte (GB), we can write

$$((\mathsf{pathname}(f) = n) \wedge (\mathsf{size}(f) = s) \wedge (s > 10^9)) \rightarrow \mathsf{bound}(n).$$

The size of the files restricts the set of objects for which information is returned, but no information is returned about the exact size of the files.

LC can also be used to state that one data source contains all the information contained in another. This sort of metadata can also be expressed in ADLIM. In fact, that is essentially what the description of a filter is. For example, from the description of `grep`, we can conclude that the input subsumes the output.

In general, any ADLIM metadata expression can be used directly as a local completeness expression for the data source that it describes, since it specifies precisely what information is represented by the data. LCW formulas can be derived from SADL sensing effects,[9] but they are quite different representationally, and information goals are still different. In ADLIM, all three are the same.

## 7 Reasoning about plans

We have implemented a planner that supports a large subset of the ADLIM language and are in the process of improving it, both in functionality and in efficiency. A discussion of the planner is beyond the scope of this paper, but here we briefly discuss how planning can be done using ADLIM, and we show a concrete example.

The main difference between ADLIM plans and plans in other languages is that actions can have inputs and outputs, which are represented as variables. For a plan to be correct, all inputs must be bound to exactly one output from an earlier action (or an existing data file). Because metadata descriptions contain universally quantified variables over universes that may never be known, the standard approach of replacing a universally quantified goal with an equivalent ground conjunction won't work; it is necessary to satisfy universally quantified goals directly with universally quantified effects; we use the same approach used in the PUCCINI planner [10].

Since a goal is a metadata expression, it has a LHS, which refers to the initial state (or earlier), and a RHS, which refers to the final state. A planner can use regression, in which the RHS is regressed backward in time until the initial state plus the LHS entail the RHS, or progression, in which the current state and the LHS are progressed forward in time until they entail the RHS. Since the LHS specifies the information that is desired, it can provide substantial guidance to the search.

### 7.1 Example

We will show an example of planning by goal regression. Let's return to the elevation map of the San Francisco Bay, discussed in Section 5. Suppose we want to produce a color image identical to this map, except that pixel values corresponding to points below sea level are blue — darker blue corresponding to greater depth. We can best describe this goal using an HSV (hue, saturation, value) representation of the color. All points should have the same value (brightness) as the original elevation map. Points above sea level should have zero saturation (gray pixels). Points below sea level should have a hue of blue and maximum saturation.

$\forall x, y$: natural, $h, s, v$: pixelValue, $elev$: real.
($x$<XMAX $\wedge$ $y$<YMAX $\wedge$
$elev$=elevation(xProj($x$),yProj($y$),XRES,YRES)) $\rightarrow$
((color($x$,$y$,$map$)=HSVcolor($h$,$s$,hProj($elev$)) $\wedge$
($elev$>0 $\rightarrow$ $s$ = 0) $\wedge$
($elev \leq 0 \rightarrow s$=MAXVALUE $\wedge$ $h$=BLUE)))

There is no need to specify the hue for pixels corresponding to points above sea level, since the hue is irrelevant if the saturation is zero. This goal can be solved by using `HSVcompose`, where the value map is the elevation map, the hue map is a solid BLUE, and the saturation map is the result of thresholding the elevation map, such that values below zero elevation correspond to MAXVALUE pixels and values above correspond to zero. If we regress the RHS through `HSVcompose`, with the I/O assignment $map = HSVout$, we get a new goal, in which the color($x$, $y$, $map$) condition is deleted (since

it is satisfied by HSVcolor), and the preconditions of HSVcolor (underlined) are added.

$\underline{\mathsf{xSize}(hue) = xs} \wedge \underline{\mathsf{xSize}(sat) = xs} \wedge$
$\underline{\mathsf{xSize}(val) = xs} \wedge \underline{\mathsf{ySize}(val) = ys} \wedge$
$\underline{\mathsf{ySize}(hue) = ys} \wedge \underline{\mathsf{ySize}(sat) = ys} \wedge$
$\underline{x < \mathsf{xSize}(hue)} \wedge \underline{y < \mathsf{ySize}(hue)} \wedge$
$\underline{\mathsf{value}(x, y, hue) = h} \wedge \underline{\mathsf{value}(x, y, sat) = s} \wedge$
$\underline{\mathsf{value}(x, y, val) = \mathsf{hProj}(elev)} \wedge$
$(elev > 0 \rightarrow s = 0) \wedge$
$(elev \leq 0 \rightarrow s = \text{MAXVALUE} \wedge h = \text{BLUE})$

We need an action to produce a threshold map corresponding to sea level. The arguments are the threshold value and the values to assign to pixels that that fall below and above the threshold.

`action` threshold(*thresh*,*below*,*above*: pixelValue)
`input` *THin*: valuemap
`output` *THout*: valuemap
`forall` *x, y*: natural, *v*: pixelValue
`effect`
xSize(*THout*) = xSize(*THin*) ∧
ySize(*THout*) = ySize(*THin*) ∧
( *x* < xSize(*THin*) ∧ *y* < ySize(*THin*) ∧
*v* = value(*x, y, THin*) )
→ ((*v*≤*thresh*→value(*x,y,THout*)=*below*)∧
(*v*>*thresh*→value(*x,y,THout*)=*above*))

We can then regress the above goal through `threshold`(hProj(0), MAXVALUE, 0), with I/O assignment *sat* = *THout*. xSize(*sat*), ySize(*sat*) and value(*x, y, sat*) are deleted, and the non-redundant preconditions of `threshold` (underlined) are added.

$\mathsf{xSize}(hue) = xs \wedge \underline{\mathsf{xSize}(THin) = xs} \wedge$
$\mathsf{xSize}(val) = xs \wedge \mathsf{ySize}(val) = ys \wedge \underline{\ }$
$\mathsf{ySize}(hue) = ys \wedge \underline{\mathsf{ySize}(THin) = ys} \wedge$
$x < \mathsf{xSize}(hue) \wedge y < \mathsf{ySize}(hue) \wedge$
$\underline{v' = \mathsf{value}(x, y, THin)} \wedge \mathsf{value}(x, y, hue) = h \wedge$
$\mathsf{value}(x, y, val) = \mathsf{hProj}(elev) \wedge$
$(elev > 0 \rightarrow \underline{v' \geq \mathsf{hProj}(0)}) \wedge$
$(elev \leq 0 \rightarrow \underline{v' \leq \mathsf{hProj}(0)} \wedge h = \text{BLUE})$

Regressing through `makeConstant`(BLUE, XMAX, YMAX) with the I/O assignment *hue* = *MCout*, the conditions value(*x, y, hue*) = *h*, *x* < xSize(*hue*), *y* < ySize(*hue*) and *h* = BLUE are satisfied, giving

$\underline{\text{XMAX}} = \mathsf{xSize}(THin) = \mathsf{xSize}(val) \wedge$
$\underline{\text{YMAX}} = \mathsf{ySize}(THin) = \mathsf{ySize}(val) \wedge$
$\mathsf{value}(x, y, val) = \mathsf{hProj}(elev) \wedge$
$\underline{x < \text{XMAX} \wedge y \leq \text{YMAX}} \wedge$
$v' = \mathsf{value}(x, y, THin) \wedge$
$(elev > 0 \rightarrow v' > \mathsf{hProj}(0)) \wedge$
$(elev \leq 0 \rightarrow v' \leq \mathsf{hProj}(0))$

Matching against the initial state with the I/O assignment *THin* = `plot` and *val* = `plot`,

$x < \text{XMAX} \wedge y < \text{YMAX} \wedge$
$\underline{elev = \mathsf{elevation}(\mathsf{xProj}(x),\mathsf{yProj}(y),\text{XRES},\text{YRES})}$
$\wedge(elev > 0 \rightarrow \underline{\mathsf{hProj}(elev)} > \mathsf{hProj}(0))$
$\wedge(elev \leq 0 \rightarrow \underline{\mathsf{hProj}(elev)} \leq \mathsf{hProj}(0))$

`action` HSVcompose(*xs*, *ys*: natural)
`input` *hue, sat, val*: valuemap
`output` *HSVout*: colormap
`precond` xSize(*hue*) = *xs* ∧ xSize(*sat*) = *xs* ∧
xSize(*val*) = *xs* ∧ ySize(*val*) = *ys* ∧
ySize(*hue*) = *ys* ∧ ySize(*sat*) = *ys*
`forall` *x, y*: natural, *h, s, v*: pixelValue
`effect`
xSize(*HSVout*) = *xs* ∧ ySize(*HSVout*) = *ys* ∧
(( *x* < *xs* ∧ *y* < *ys* ∧ value(*x, y, hue*) = *h* ∧
value(*x, y, sat*) = *s* ∧ value(*x, y, val*) = *v*)
→ color(*x,y,HSVout*)=HSVcolor(*h,s,v*))

Figure 2: `HSVcompose` composes three monochrome images into a color image. This is essentially the same as the `compose` action in Figure 1, except that compose uses an RGB (red, green, blue) color model, whereas this action use an HSV (hue, saturation, value) representation. The inputs are three *valuemaps*, which are essentially monochrome images containing the hues, saturations and values that are to be combined. The output is a *colormap*, whose pixel values are numbers that can be also represented as RGB or HSV triples. The function HSVcolor returns the corresponding color value for a given h,s,v triple. The sole precondition is that the three input images have the same dimensions, and the effect is to produce the corresponding output, also with the same dimensions. The action quantifies over pixels in the images. The variables $x$ and $y$ are natural numbers, and $h$, $s$ and $v$ are the values of pixels in the grayscale images.

The first three terms are entailed by the LHS. The rest follows from the fact that hProj is an increasing linear function.

## 8 Conclusions

We discussed a practical planner-based approach to data management that facilitates both data processing and tracking. This approach removes the need for software designers or mission operators to custom design solutions to every type of data request, allowing them instead to focus on providing models of available commands and information resources. We illustrated the approach with the ADLIM language and showed how to reason about data-processing plans in this language. We also showed how ADLIM can be used for information gathering as well as data processing, and demonstrated that ADLIM metadata formulas subsume Local Completeness. This work builds on ideas from previous languages, particularly SADL [10] and the LC representation from the Razor system [7]. The LHS of information goals is similar to **initially** goals in SADL, which are used to refer to information about the initial state.

We have implemented a simple planner that supports an earlier version of ADLIM, but much work remains. Although ADLIM is simple language, the sim-

plest we could think of that is capable of representing data management domains, metadata can be quite complex, since they describe data that in turn describe the world. Thus, the meta problem of figuring out how to represent the metadata for a particular file or action can be a challenge, involving complex spatio-temporal representation.

We are exploring ways to reason about plan quality, which we define as a numeric function of attributes like data volume and image compression.

## References


[1] S. Chien, F. Fisher, E. Lo, H. Mortensen, and R. Greeley. Using artificial intelligence planning to automate science data analysis for large image database. In *Proc. 1997 Conference on Knowledge Discovery and Data Mining*, August 1997.

[2] O. Duschka, M. Genesereth, and A. Levy. Recursive query plans for data integration. *Journal of Logic Programming, special issue on Logic Based Heterogeneous-Information Systems*, 2000.

[3] O. Etzioni, K. Golden, and D. Weld. Sound and efficient closed-world reasoning for planning. *J. Artificial Intelligence*, 89(1–2):113–148, January 1997.

[4] O. Etzioni and D. Weld. A softbot-based interface to the Internet. *C. ACM*, 37(7):72–6, 1994.

[5] Oren Etzioni, Keith Golden, and Dan Weld. Tractable closed-world reasoning with updates. In *Proc. 4th Int. Conf. Principles of Knowledge Representation and Reasoning*, pages 178–189, 1994.

[6] Oren Etzioni, Steve Hanks, Daniel Weld, Denise Draper, Neal Lesh, and Mike Williamson. An approach to planning with incomplete information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 115–125, 1992.

[7] M. Friedman and D. Weld. Efficiently executing information-gathering plans. In *Proc. 15th Int. Joint Conf. AI*, pages 785–91, 1997.

[8] K. Golden and D. Weld. Representing sensing actions: The middle ground revisited. In *Proc. 5th Int. Conf. Principles of Knowledge Representation and Reasoning*, pages 174–185, 1996.

[9] Keith Golden. *Planning and Knowledge Representation for Softbots.* PhD thesis, University of Washington, 1997. Available as UW CSE Tech Report 97-11-05.

[10] Keith Golden. Leap before you look: Information gathering in the PUCCINI planner. In *Proc. 4th Intl. Conf. AI Planning Systems*, 1998.

[11] Z. Ives, A. Levy, D. Weld, D. Forescu, and M. Friedman. Adaptive query processing for internet applications. *Data Engineering Bulletin*, 23(2), 2000.

[12] Craig Knoblock. Building a planner for information gathering: A report from the trenches. In *Proc. 3rd Intl. Conf. AI Planning Systems*, 1996.

[13] C. Kwok and D. Weld. Planning to gather information. In *Proc. 13th Nat. Conf. AI*, 1996.

[14] A. L. Lansky and A. G. Philpot. AI-based planning for data analysis tasks. In *Proceedings of the Ninth IEEE Conference on Artificial Intelligence for Applications (CAIA-93)*, 1993.

[15] A. Levy. Obtaining complete answers from incomplete databases. In *Proc. 22nd VLDB Conf.*, 1996.

[16] Alon Y. Levy, A. Rajaraman, and Joann J. Ordille. Query answering algorithms for information agents. In *Proc. 13th Nat. Conf. AI*, 1996.

[17] Sheila A. McIlraith and Richard B. Scherl. What sensing tells us: Towards a formal theory of testing for dynamical systems. In *AAAI/IAAI*, pages 483–490, 2000.

[18] R. Moore. A Formal Theory of Knowledge and Action. In J. Hobbs and R. Moore, editors, *Formal Theories of the Commonsense World.* Ablex, 1985.